Session 16
Software Testing Metrics

xfzhang@suda.edu.cn

Review 软件缺陷报告

- 软件测试执行与跟踪
- 软件缺陷的描述
  - 软件缺陷的生命周期
  - 严重性和优先级
  - 缺陷的其它属性
  - 完整的缺陷信息
  - 缺陷描述的基本要求
  - 缺陷报告的示例
- 软件缺陷跟踪和分析

书写软件测试和质量分析报告之前

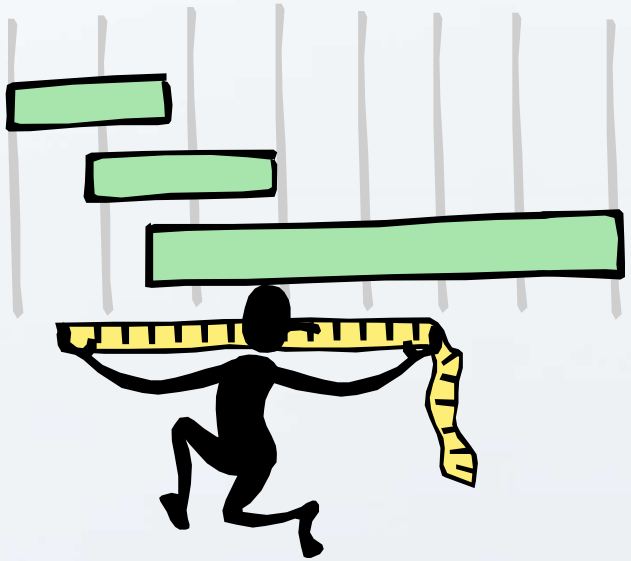
- 是否完成了测试计划所要求的各项测试内容？
- 测试用例是否经过开发人员、产品经理的严格评审？
- 需要执行的测试用例是否百分之百地完成了？
- 单元测试的代码行覆盖率是否达到所设定的目标？
- 集成测试是否全面验证了所有接口及其参数？
- 系统测试是否包含了性能、兼容性、安全性、恢复性等各项测试？如果执行了，又是怎么进行的、结果如何？
- 所有严重的Bug都修正了？

测试和软件质量分析报告

- 什么是质量度量
- 基于覆盖的质量评估
- 基于缺陷分析的质量评估
- 测试报告的具体内容

软件产品的质量度量

- 软件度量及其过程
- 软件质量的度量
- 质量度量的统计方法



软件度量及其过程

软件度量是对软件所包含的各种属性的量化表示

定性 → 定量

软件度量可以帮助我们深入了解软件过程和产品的衡量指标，使组织能够更好地做出决策以达成目标：

- 用数据指标表明验收标准；
- 监控项目进度和预见风险；
- 分配资源时进行量化均衡；
- 预计和控制产品的过程、成本和质量。

eg.学分制

度量概念

测量（Measurement）：确定一个测量的行为

度量（Metric）：某个给定属性的度的一个定量测量

指标 (Indicator)：具体测量的属性及其给定值，或组合值

举例

measure: 文档页数, 发现错误数，每个人的准备时间

metrics: preparation rate = 总的准备时间 / 文档页数

fault density = 错误数 / 文档页数

indicator:

准备程度：由preparation rate这一个metric表示

查错有效性：由fault density这一个metric表示

正常程度：由prep rate和fault density两个metrics组成的二维空间里去判断

软件度量的内容

- 规模度量：代码行数，功能点和对象点等
- 复杂度度量：软件结构复杂度指标
- 缺陷度量：帮助确定产品缺陷变化的状态，并指示修复缺陷活动所需的工作量，分析产品缺陷分布的情况
- 工作量度量
- 进度度量
- 生产率度量：代码行数 / 人·月，测试用例数/人·日；
- 风险度量：“风险发生的概率”和“风险发生后所带来的损失”
- …

Types of Test Metrics

- There are various types of test metrics that can be gathered in a software development project.
- Most projects need measures of quality, resources, time, and size.
  - Size measurements
  - Complexity measurements
  - Metrics unique to test

Size Measurements

- Size is a fundamental metric.
- Most of the metrics that are gathered are normalized by size metrics. This provides a size independent analysis of a software project.
- The size of a software can be calculated in the following ways:
  - Lines of Code (LOC)
  - Function Points (FPs)
  - Tokens

Size Measurements

- Lines of Code (LOC)
  - depends on the language and individual style

```
for ( i=0; i<n; i++)
{
  printf ( “%d\n”, i);
}

for ( i=0; i<n; i++, printf ( “%d\n”, i) );
```

- Function Points (FPs)
  - Independent of language and LOC
  - Can be measured early

Size Measurements

- Tokens
  - Tokens are operators and operands used in a software coding project.
  - Total number of operator tokens used, N1
  - Total number of operand tokens used, N2
  - Number of unique operator tokens, n1
  - Number of unique operand tokens, n2

- The software program size N, can be measured on the basis of tokens by:

$$N=N1+N2=n1\ \log_2 n1+n2\ \log_2 n2$$

Complexity Measurements

- Complexity metrics is a component-level design metric.
- Component-level design metrics help focus on the internal characteristics of the software at the component level.
- These metrics provide critical information such as the reliability and maintainability of the software systems.
- Cyclomatic complexity

有效软件度量的属性

- 简单的，可计算的
- 经验和直觉上有说服力
- 一致的和客观的
- 在其单位和维度的使用上是有意义的
- 编程语言独立的
- 质量反馈的有效机制
- …

软件度量的过程

- **识别目标。**分析出度量的工作目标和列表，并由管理者审核确认
- **定义度量过程。**定义其收集要素、收集过程、分析、反馈过程、IT支持体系，为具体的收集活动、分析、反馈活动和IT设备、工具开发提供指导。
- **搜集数据。**应用IT支持工具进行数据收集工作，并按指定的方式审查和存储。
- **数据分析与反馈。**根据数据收集结果，按照已定义的分析方法进行数据分析，完成规定格式的图表，进行反馈。
- **过程改进。**根据度量的分析报告，管理者基于度量数据做出决策。

软件质量的度量

基于质量模型，带加权因子的回归公式来度量质量

$$Mi = c1 \times f1 + c2 \times f2 + \ldots + cn \times fn$$

Mi是一个软件质量因素

fn是影响质量因素的度量值：软件复杂度度量、缺陷度量和规模度量 …

cn是加权因子

质量度量的统计方法

量化评估　12条错误原因

- 说明不完整或说明错误 (IES)
- 与客户交流不够所产生的误解 (MCC)
- 故意与说明偏离 (IDS)
- 违反编程标准 (VPS)
- 数据表示有错 (EDR)
- 模块接口不一致 (IMI)
- 设计逻辑有错 (EDL)
- 不完整或错误的测试 (IET)
- 不准确或不完整的文档 (IID)
- 将设计翻译成程序设计语言中的错误 (PLT)
- 不清晰或不一致的人机界面 (HCI)
- 杂项 (MIS)

质量度量的统计方法 （2）

总计(E_i)			严重(S_i)		一般(M_i)		微小(T_i)	
错误	数量	百分比	数量	百分比	数量	百分比	数量	百分比
IES	296	**22.3%**	55	**28.2%**	95	18.6%	146	23.4%
MCC	204	**15.3%**	18	9.2%	87	17.0%	99	15.9%
IDS	64	4.8%	2	1.0%	31	6.1%	31	5.0%
VPS	34	2.6%	1	0.5%	19	3.7%	14	2.2%
EDR	182	**13.7%**	38	**19.5%**	90	17.6%	54	8.7%
IMI	82	6.2%	14	7.2%	21	4.1%	47	7.5%
EDL	64	4.8%	20	**10.3%**	17	3.3%	27	4.3%
IET	140	**10.5%**	17	8.7%	51	10.0%	72	11.6%
IID	54	4.1%	3	1.5%	28	5.5%	23	3.7%
PLT	87	6.5%	22	**11.3%**	26	5.1%	39	6.3%
HCI	42	3.2%	4	2.1%	27	5.3%	11	1.8%
MIS	81	6.1%	1	0.5%	20	3.9%	60	9.6%
总计	1330	100%	195	100%	512	100%	623	100%

质量度量计算

- 阶段错误度量　区分错误的严重程度

$$PI_i = W_s\,(S_i/E_i) + W_m(M_i/E_i) + W_i\,(T_i/E_i)$$

- 总体质量度量

$$E_p = \Sigma\,(i * PI_i)/P_s$$

i = 1, 2, 3, 4, 5 代表需求分析、设计、编程、测试、发布
(或 i= 1, 2, 5, 10, 100)

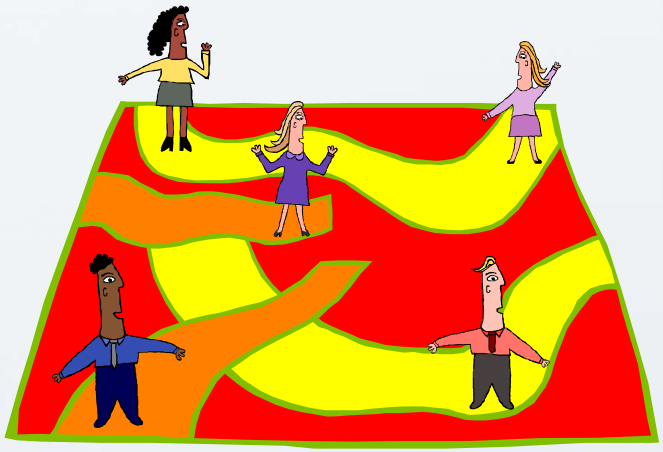
W_s, W_m, W_i = 0.6, 0.3, 0.1

测试和软件质量分析报告

- 什么是质量度量
- 基于覆盖的质量评估
- 基于缺陷分析的质量评估
- 测试报告的具体内容

评估系统测试的覆盖程度

- 基于需求的测试覆盖评估
- 基于代码的测试覆盖评估



测试的评估

软件测试评估主要有两个目的

- 量化测试进程，判断测试进行的状态和进度
- 为测试或质量分析报告生成所需的量化数据，如缺陷清除率、测试覆盖率等

基于需求的测试覆盖评估

假定 T_x已执行的测试过程数或测试用例数，R_{ft}是测试需求的总数：

$$已执行的测试覆盖 = T_x / R_{ft}$$

假定 T_s是已执行的完全成功、没有缺陷的测试过程数或测试用例数：

$$成功的测试覆盖 = T_s / R_{ft}$$

基于代码的测试覆盖评估

基于代码的测试覆盖评测是对被测试的程序代码语句、路径或条件的覆盖率分析。

基于代码的测试覆盖通过以下公式计算：

$$\text{已执行的测试覆盖} = T_c / T_{nc}$$

其中T_c是用代码语句、条件分支、代码路径、数据状态判定点等已执行项目数，

T_{nc}（Total number of items in the code）是代码中的项目总数。

测试和软件质量分析报告

- 什么是质量度量
- 基于覆盖的质量评估
- 基于缺陷分析的质量评估
- 测试报告的具体内容

缺陷评测的基线

为软件产品的质量设置起点，在基线的基础上再设置新的目标，作为对系统评估是否通过的标准

条目	目标	低水平
缺陷清除效率	>95%	<70%
原有缺陷密度	每个功能点 <4	每个功能点 >7
超出风险之外的成本	0%	>=10%
全部程序文档	每个功能点页数 <3	每个功能点页数 >6
员工离职率	每年1 to 3%	每年>5%

基于缺陷清除率的估算方法

F为描述软件规模用的功能点；D1为在软件开发过程中发现的所有缺陷数；D2为软件发布后发现的缺陷数；D为发现的总缺陷数。D=D1+D2

- 缺陷注入率(缺陷密度) = D/F
- 整体缺陷清除率=D1/D

缺陷源	潜在缺陷	清除效率(%)	遗留的缺陷
需求报告	1.00	77	0.23
设计	1.25	85	0.19
编码	1.75	95	0.09
文档	0.60	80	0.12
错误修改	0.40	70	0.12
合计	5.00	85	0.75

经典的种子公式

$$\frac{\text{已测试出的种子Bug (s)}}{\text{所有的种子Bug (S)}} = \frac{\text{已测试出的非种子Bug (n)}}{\text{全部的非种子Bug (N)}}$$

$$N = S * n / s$$

其中n是所进行实际测试时发现的Bug总数。如果 n = N, 可以推测为所有的Bug已找出来，说明做的测试足够充分。

问题:

- 种子bug的代表性
- 人为设置程序的bug的困难
- 缺陷相互之间可能存在相互影响或关联-**Bug的非单调性**

	Correct Program $\mathcal{P}$	$\mathcal{P}$ with fault f_1	$\mathcal{P}$ with fault f_2	$\mathcal{P}$ with fault f_1 and f_2

	read(a,b);
	x=a;
	y=b;
	if((x+y) == 6)
	print(6);
	else
	print(-1);
		read(a,b);
		x=a + 3;
		y=b;
		if((x+y) == 6)
		print(6);
		else
		print(-1);
			read(a,b);
			x=a;
			y=b + 3;
			if((x+y) == 6)
			print(6);
			else
			print(-1);
				read(a,b);
				x=a + 3;
				y=b + 3;
				if((x+y) == 6)
				print(6);
				else
				print(-1);

Test t_1 Input: a = 0 b = 0	output: -1	output: -1	output: -1	output: 6
		Pass	***Pass***	***Fail***

图 2.4: Bug 相长干涉示例

	Correct Program $\mathcal{P}$	$\mathcal{P}$ with fault f_1	$\mathcal{P}$ with fault f_2	$\mathcal{P}$ with fault f_1 and f_2

	read(a);
	x=a;
	y=x - 3;
	print(y);
		read(a);
		x=a + 1;
		y=x - 3;
		print(y);
			read(a);
			x=a;
			y=x - 4;
			print(y);
				read(a);
				x=a + 1;
				y=x - 4;
				print(y);

Test t_1 Input: a = 5	output: 2	output: 3	output: 1	output: 2
		Fail	***Fail***	***Pass***

图 2.5: Bug 相消干涉示例

	Correct Program $\mathcal{P}$	$\mathcal{P}$ with fault f_1	$\mathcal{P}$ with fault f_2	$\mathcal{P}$ with fault f_1 and f_2

	read(a,b);
	x=a;
	y=b;
	result=0;
	if((x+y) == -7)
	result = 1;
	else
	result = 2;
	print(result);
		read(a,b);
		x=a - 3;
		y=b;
		result=0;
		if((x+y) == -7)
		result = 1;
		else
		result = 2;
		print(result);
			read(a,b);
			x=a;
			y=b - 4;
			result=0;
			if((x+y) == -7)
			result = 1;
			else
			result = 2;
			print(result);
				read(a,b);
				x=a - 3;
				y=b - 4;
				result=0;
				if((x+y) == -7)
				result = 1;
				else
				result = 2;
				print(result);

Test t_1 Input: a = 0 b = 0	output: 2	output: 2	output: 2	output: 1
		Pass	***Pass***	***Fail***
Test t_2 Input: a = -4 b = 0	output: 2	output: 1	output: 2	output: 2
		Fail	***Pass***	***Pass***

图 2.6: Bug 干涉示例

Bug的非单调性

变异测试（Mutation Testing）

变异测试是一种 fault-based 的软件测试技术。

```
if(a && b)
        c = 1;
else c = 0;
```

```
if(a || b)
        c = 1;
else c = 0;
```

为了杀死这个变异，需要满足以下条件：

（1）测试数据必须对变异和原始程序引起的不同状态覆盖。如：a=1, b=0可以达到目的。

（2）c的值应该传播到程序输出，并被测试检查。

弱变异覆盖需满足(1)，

强变异覆盖需满足(1)(2)。

VS. Decision Coverage

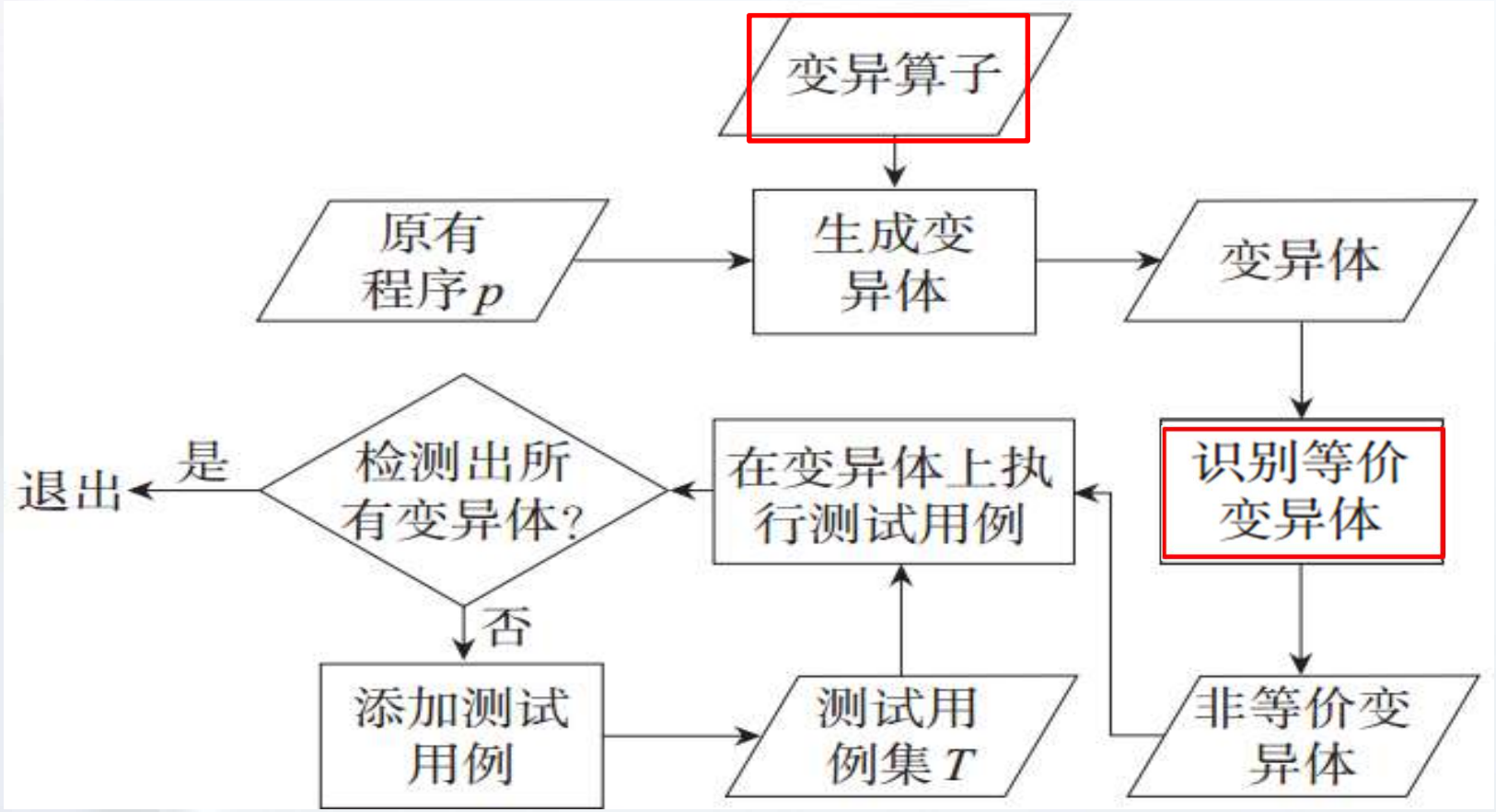
变异测试（Mutation Testing）

- 通过采用变异缺陷来模拟被测软件的真实缺陷，从而对研究人员提出的测试方法的有效性进行辅助评估。
- 变异测试旨在找出有效的测试用例，发现程序中真正的错误。传统的变异测试旨在寻找这些错误的子集，能尽量充分地近似描述这些BUG。
- 变异测试则提供了基于缺陷的对测试充分性进行度量的角度，针对测试用例集的充分性进行评估和改进。

变异测试（Mutation Testing）

- 这个理论基于两条假设：
  - Competent Programmer Hypothesis(CPH)
    - 假设编程人员是有能力的，他们尽力去更好地开发程序，达到正确可行的结果，而不是搞破坏。
  - Coupling Effect(CE)
    - 关注在变异测试中错误的类别。
    - 复杂变异体往往是由诸多简单变异体组合而成。

变异测试（Mutation Testing）



变异测试的流程

变异测试（Mutation Testing）

- 变异算子
  - 在符合语法规则前提下， 变异算子定义了从原有程序生成差别极小程序（即变异体） 的转换规则。

程序 p	变异体 p'
…	…
if $(a+b>c)$	if $(a-b>c)$
return true;	return true;
…	…

变异测试（Mutation Testing）

- 1987年针对Fortran77首次定义了 22种变异算子

序号	变异算子	描述
1	AAR	用一数组引用替代另一数组引用
2	ABS	插入绝对值符号
3	ACR	用数组引用替代常量
4	AOR	算术运算符替代
5	ASR	用数组引用替代变量
6	CAR	用常量替代数组引用
7	CNR	数组名替代
8	CRP	常量替代
9	CSR	用常量替代变量
10	DER	DO语句修改
11	DSA	DATA语句修改
12	GLR	GOTO标签替代
13	LCR	逻辑运算符替代
14	ROR	关系运算符替代
15	RSR	RETURN语句替代
16	SAN	语句分析
17	SAR	用变量替代数组引用
18	SCR	用变量替代常量
19	SDL	语句删除
20	SRC	源常量替代
21	SVR	变量替代
22	UOI	插入一元操作符

变异测试（Mutation Testing）

- 常见变异算子

- Statement deletion
- Statement duplication or insertion, e.g. `goto fail;` [16]
- Replacement of boolean subexpressions with *true* and *false*
- Replacement of some arithmetic operations with others, e.g. `+` with `*`, `-` with `/`
- Replacement of some boolean relations with others, e.g. `>` with `>=`, `==` and `<=`
- Replacement of variables with others from the same scope (variable types must be compatible)

变异测试（Mutation Testing）

- 在完成变异算子设计后，通过在原有被测程序上执行变异算子可以生成大量变异体M，在变异测试中，变异体一般被视为含缺陷程序。
- 根据执行变异算子的次数，可以将变异体分为一阶变异体和高阶变异体
  - （一阶变异体） 在原有程序 p 上执行单一变异算子并形成变异体 p′，则称p′为p的一阶变异体。
  - （高阶变异体） 在原有程序 p 上依次执行多次变异算子并形成变异体 p′，则称 p′ 为 p 的高阶变异体。若在 p 上依次执行 k 次变异算子并形成变异体 p′， 则称 p′ 为 p 的 k 阶变异体。

变异测试（Mutation Testing）

- 高阶变异体实例

```
输入：a,x,y
1. z=x;
2. z=z+y;
3. if (a>0)
4.     return z;
5. else
6.     return 2*x+z;
```

测试用例	$a=1$	$a=-1$
原有程序	$x+y$	$3x+y$
变异体1		
变异体2		
变异体12		

其中：

变异体1（一阶变异体）：

将第一行变异为 $z=++x$

变异体2（一阶变异体）：

将第二行变异为 $z=z+--y$

变异体12（二阶变异体）：

合并变异体1和变异体2

两个测试用例：

（1）$a=1$　　（2）$a=-1$

变异测试（Mutation Testing）

- 高阶变异体实例

```
输入：a,x,y
1. z=x;
2. z=z+y;
3. if (a>0)
4.     return z;
5. else
6.     return 2*x+z;
```

测试用例	$a=1$	$a=-1$
原有程序	$x+y$	$3x+y$
变异体1	$x+y+1$	$3x+y+3$
变异体2	$x+y-1$	$3x+y-1$
变异体12	$x+y$	$3x+y+2$

其中：

变异体1(一阶变异体)：

将第一行变异为 $z=++x$

变异体2(一阶变异体)：

将第二行变异为 $z=z+--y$

变异体12(二阶变异体)：

合并变异体1和变异体2

两个测试用例：

(1)a=1　　(2)a=−1

变异测试（Mutation Testing）

- 可杀除变异体
  - 若存在测试用例 t，在变异体 p′ 和原有程序 p 上的执行结果不一致， 则称该变异体 p′ 相对于测试用例集 T 是可杀除变异体。

mutation score = number of mutants killed / total number of mutants

变异测试（Mutation Testing）

- 可存活变异体
  - 若不存在任何测试用例t， 在变异体 p′ 和原有程序 p 上的执行结果不一致， 则称该变异体 p′ 相对于测试用例集 T 是可存活变异体。
  - 一部分可存活变异体通过设计新的测试用例可以转化成可杀除变异体，剩余的可存活变异体则可能是等价变异体。

变异测试（Mutation Testing）

- 等价变异体
  - 若变异体 p′ 与原有程序 p 在语法上存在差异，但在语义上与 p 保持一致， 则称p′ 是 p 的等价变异体。

程序 p	变异体 p'
for (int $i=0;i<10;i++$){	for (int $i=0;i!=10;i++$){
$sum+=a[i];$	$sum+=a[i];$
}	}

变异测试（Mutation Testing）

- 等价变异体检测
  - 是一个不可判定问题，因此需要测试人员借助手工方式予以完成。
  - 等价变异体在语法层次上有微小的差别，但是在语义层次上是一致的。
  - 有研究人员发现，在生成的大量变异体中，等价变异体所占比例一般介于10%~40%。

变异测试（Mutation Testing）

- 变异体选择优化
  - 变异体选择优化策略主要关注如何从生成的大量变异体中选择出典型变异体。
  - 随机选择法
  - 聚类选择法
  - 变异算子选择法
  - 高阶变异体优化法

变异测试（Mutation Testing）

- 变异体选择优化--随机选择法
  - 尝试从生成的大量变异体中随机选择出部分变异体。
  - 首先通过执行变异算子生成大量变异体 M ；
  - 然后定义选择比例 x ；
  - 最后从变异体 M 中随机选择出 |M| * x% 的变异体， 剩余未被选择的变异体则被丢弃。

变异测试（Mutation Testing）

- 变异体选择优化--聚类选择法
  - 首先对被测程序p 应用变异算子生成所有的一阶变异体；
  - 然后选择某一聚类算法根据测试用例的检测能力对所有变异体进行聚类分析，使得每个聚类内的变异体可以被相似测试用例检测到；
  - 最后从每个聚类中选择出典型变异体， 而其他变异体则被丢弃。

变异测试（Mutation Testing）

- 变异体选择优化--变异算子选择法
  - 从变异算子选择角度出发， 希望在不影响变异评分的前提下， 通过对变异算子进行约简来大规模缩小变异体数量， 从而减小变异测试和分析开销。
  - 结果表明， 变异算子选择法相对于随机选择法来说并不存在明显优势

变异测试（Mutation Testing）

- 变异体选择优化--高阶变异体优化法
  - 高阶变异体优化法基于如下推测：

    （1）执行一个k 阶变异体相当于一次执行 k 个一阶变异体；

    （2） 高阶变异体中等价变异体的出现概率较小。
  - 实证研究表明， 采用二阶变异体可以有效减少50%的测试开销， 但却不会显著降低测试的有效性。

变异测试（Mutation Testing）

- 应用在Java上的工具有：
  - MuJava
  - MuClipse
  - PiTest

Pit Test Coverage Report

Package Summary

com.eason.tool.pitest.demo

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	75%	3/4	83%	5/6	83%	5/6

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
AgeUtils.java	75%	3/4	83%	5/6	83%	5/6

Report generated by PIT 1.6.4

PiTest的index页面

```
9  public class AgeUtils {
10     public static boolean isChild(int age) {
11 4       if (age >= 2 && age < 18) {
12 1           return true;
13         }
14 1       return false;
15     }
16 }
```

Mutations

```
1. changed conditional boundary → KILLED
2. changed conditional boundary → SURVIVED
3. negated conditional → KILLED
4. negated conditional → KILLED
```

11 1. changed conditional boundary → KILLED 2. changed conditional boundary → SURVIVED 3. negated conditional → KILLED 4. negated conditional → KILLED

12 1. replaced boolean return with false for com/eason/tool/pitest/demo/AgeUtils::isChild → KILLED

14 1. replaced boolean return with true for com/eason/tool/pitest/demo/AgeUtils::isChild → KILLED

Active mutators

- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Tests examined

- com.eason.tool.pitest.demo.AgeUtilsTest (362 ms)

Report generated by PIT 1.6.4

测试详情界面

Metrics Unique to Test--DRE

- The common metrics unique to testing include:
  - Defect Removal Efficiency (DRE): It is the measure of defects before the delivery of software.
  - DRE = (Total Number of defects found during development (before Delivery)) / (Total Number of defects found during development (before Delivery) + Total Number of Defects Found after Delivery)
  - DRE indicates the defect filtering ability of the testing processes.

Metrics Unique to Test--DD

- The common metrics unique to testing include:
  - Defect Density (DD): It is the measure of all the found defects per size of software entity being measured.
  - DD = Number of Known Defects / Size (LOC, FP or Tokens)
  - The relative number of defects helps the management to focus on components for additional inspection, testing, re-engineering, or replacement.

Exercise 1

Stage Discovered	Defect Count
Requirements Review	50
Design review	30
Code review	20
Testing	25
After Delivery	10

The total size of the software is 483 FP.

Calculate the DRE and DD for the project and help the management analyze data.

Exercise 1

DRE = (Total Number of defects found during development (before Delivery)) / (Total Number of defects found during development (before Delivery) + Total Number of Defects Found after Delivery)

Total number of defects found before delivery or N1 = Defects found in Requirements Review + Defects found in Design review + Defects found in Code review + Defects found in System Testing

Therefore, N1 = 50 + 30 + 20 + 25 = 125 defects

Total number of defects found after delivery or N2 = Defects found in acceptance testing

Therefore, N2 = 10

DRE = N1/ (N1 + N2) = 125 / (125 + 10) = 125/135 = 0.92

Exercise 1

DD is the defect density in a software project. This provides a normalized view of the software.

DD = Number of Known Defects / Size (in LOC or FP)

The total number of known defects

N = 50 + 30 + 20 + 25 + 10 = 135

DD = 135 / 483 FP = 0.27 defects per Function Point.

Metrics Unique to Test-MTTF

1 MTTR——Mean Time To Repair，即平均恢复时间。就是从出现故障到恢复中间的这段时间。MTTR越短表示易恢复性越好。

2. MTTF——Mean Time To Failure，即平均无故障时间/故障前平均时间。系统的可靠性越高，平均无故障时间越长。

3. MTBF——Mean Time Between Failure，即平均失效间隔。包括故障时间以及检测和维护设备的时间。MTBF = MTTF + MTTR。因为MTTR通常远小于MTTF，所以MTBF近似等于MTTF，通常由MTTF替代。

Metrics Unique to Test-MTTF

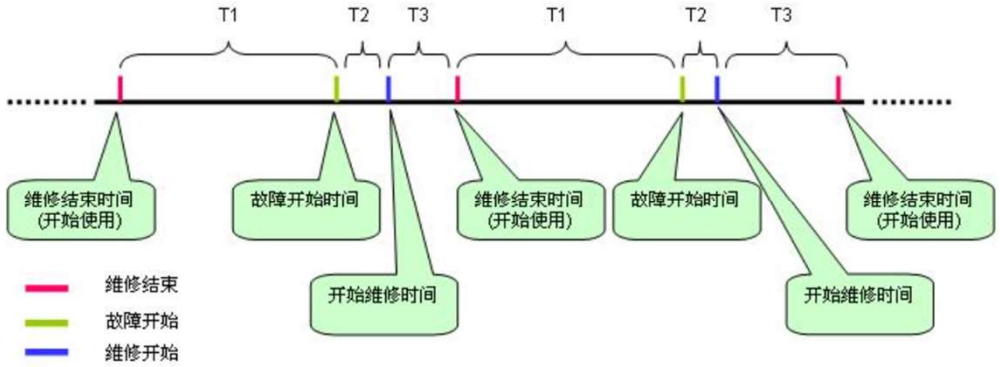
MTTF (Mean Time To Failure，平均无故障时间)，指系统无故障运行的平均时间，取所有从系统开始正常运行到发生故障之间的时间段的平均值。 MTTF =∑T1/ N

MTTR (Mean Time To Repair，平均修复时间)，指系统从发生故障到维修结束之间的时间段的平均值。MTTR =∑(T2+T3)/ N

MTBF (Mean Time Between Failure，平均失效间隔)，指系统两次故障发生时间之间的时间段的平均值。 MTBF =∑(T2+T3+T1)/ N

很明显:MTBF= MTTF+ MTTR

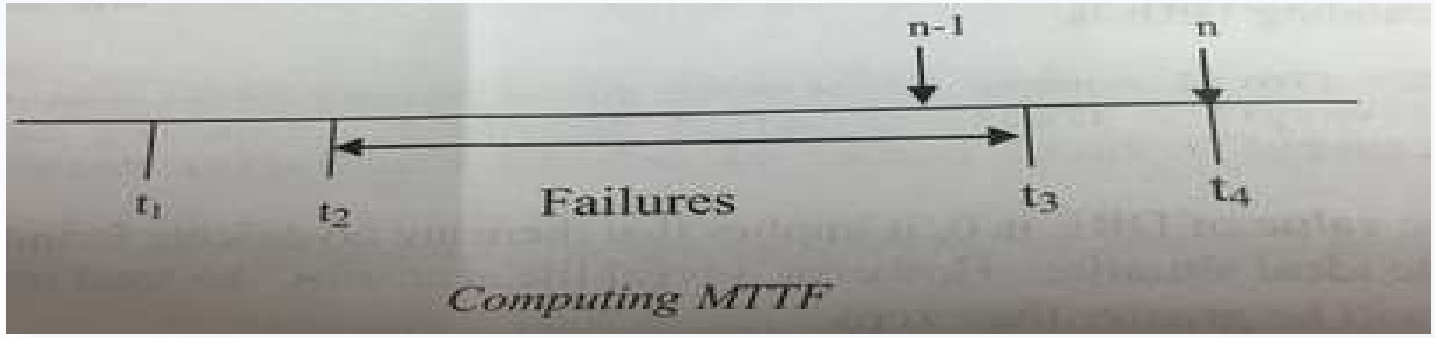
图解 MTTR、MTTF、MTBF



Metrics Unique to Test--MTTF

- The common metrics unique to testing include:
  - Mean Time to Failure (MTTF): It is a measure of reliability. It provides the average time from one failure to the next during a test operation on a software project.

Metrics Unique to Test-MTTF



2和3之间没有缺陷，就往前找最近的出错的1和往后找最近的出错的4去cover2和3；间隔：缺陷数-1

1 Define the period over which the MTTF is to be calculated as starting time, t2 and ending time, t3.

2 Locate the latest failure earlier than t2 and find the time, t1, at which this failure occurred.

3 Locate the earliest failure later than t3 and find the time, t4, at which it occurred, meaning the two failures nearest to the period but outside the period.

4 Count the number of failures, including these two and all failures in between.

5 Calculate the MTTF for the period t2 to t3 by (t4-t1)/(n-1)

Metrics Unique to Test-MTTF

Month	Dates
May	2nd
June	5th
July	4th 28th
August	4th, 28th

Failures recorded over six months

Calculate the MTTF for June, July and August.

Calculate the MTTF over these four months.

7月：（8.4-6.5）/（4-1）
8月：（8.28-8.4）/（2-1）
过去四个月：（8.28-5.2）/（6-1）

缺陷损耗的估算方法

- 缺陷潜伏期，通常也称为阶段潜伏期。缺陷潜伏期是一种特殊类型的缺陷分布度量。
- 在实际测试工作中，发现缺陷的时间越晚，这个缺陷所带来的损害就越大，修复这个缺陷所耗费的成本就越多。
- 缺陷损耗对缺陷的发现过程有效性和修复软件缺陷所耗费的成本等进行评测。

表 5-1　　一个项目的缺陷潜伏期的度量

缺陷造成阶段	发现阶段									
	需求	总体设计	详细设计	编码	单元测试	集成测试	系统测试	验收测试	试运行产品	发布产品
需求	0	1	2	3	4	5	6	7	8	9
总体设计		0	1	2	3	4	5	6	7	8
详细设计			0	1	2	3	4	5	6	7
编码				0	1	2	3	4	5	6
总计										

表 5-2　　一个项目的缺陷分布情况

缺陷造成阶段	发现阶段										缺陷总量
	需求	总体设计	详细设计	编码	单元测试	集成测试	系统测试	验收测试	试运行产品	发布产品	
需求	0	8	4	1	0	0	5	6	2	1	27
总体设计		0	9	3	0	1	3	1	2	1	20
详细设计			0	15	3	4	0	0	1	8	31
编码				0	62	16	6	2	3	20	109
总计	0	8	13	19	65	21	14	9	8	30	187

缺陷损耗的估算方法

缺陷损耗是使用阶段潜伏期和缺陷分布来度量缺陷消除活动的有效性的一种度量。

$$\text{缺陷消耗} = \frac{\text{缺陷数量} \times \text{发现的阶段潜伏期加权值}}{\text{缺陷总量}}$$

表 5-3　　一个项目的各个缺陷损耗值

缺陷造成阶段	发现阶段										缺陷损耗
	需求	总体设计	详细设计	编码	单元测试	集成测试	系统测试	验收测试	试运行产品	发布产品	
需求	0	8	8	3	0	0	30	42	16	9	4.3
总体设计		0	9	6	0	4	15	6	14	8	*3.1*
详细设计			0	15	6	12	0	0	6	56	*3.06*
编码				0	62	32	18	8	15	120	*2.34*
总计											

缺陷损耗的估算方法

- 一般而言，缺陷损耗的数值越低，说明缺陷的发现过程越有效（最理想的数值应该为1）。
- 用缺陷损耗来度量测试有效性的长期趋势（递减）时，它就会显示出自己的价值。

测试和软件质量分析报告

- 什么是质量度量
- 基于覆盖的质量评估
- 基于缺陷分析的质量评估
- 测试报告的具体内容

测试报告的具体内容

- 测试总结报告的目的是总结测试活动的结果，并根据这些结果对测试进行评价。
- 这种报告是测试人员对测试工作进行总结，并识别出软件的局限性和发生失效的可能性。
- 测试总结报告是测试计划的扩展，起着对测试计划“封闭回路”的作用。

测试报告的具体内容

在国家标准GB/T 17544－1998（附录C）对测试报告有了具体要求，对测试纪录、结果如实汇总分析，报告出

- 产品标识；
- 用于测试的计算机系统；
- 使用的文档及其标识；
- 产品描述、用户文档、程序和数据的测试结果；
- 与要求不符的清单；
- 针对建议的要求不符的清单，产品未作符合性测试的说明；
- 测试结束日期。

IEEE 标准 829－1998 软件测试文档编制标准

测试总结报告模板

目录

Bug report

Testing metrics

测试总结报告模板

测试和软件质量分析报告

- 什么是质量度量
- 基于覆盖的质量评估
- 基于缺陷分析的质量评估
- 测试报告的具体内容